JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 73–91 (1999)

Research

An Evaluation of the Cognitive Processes of Programmers Engaged in Software Debugging

JOANNE E. HALE*, SHANE SHARPE and DAVID P. HALE

Culverhouse College of Commerce and Business Administration, University of Alabama, Tuscaloosa AL 35487–0226, U.S.A.

SUMMARY

This study empirically evaluates Hale and Haworth's cognitive processes model for programmers engaged in software debugging. They claim that across all levels of experience, software programmers engage in the same activities and follow a similar pattern of activities directed by search strategies. Verbal protocol analysis of data gathered during a staged problem-solving episode (e.g., fixing a hidden bug) was used to evaluate Hale and Haworth's hypothesized debugging tasks, as well as the hypothesized sequence in which the tasks are performed. Empirical support for Hale and Haworth's model was strong. Specifically, the set of hypothesized debugging tasks is both accurate and complete. Further, the model effectively represents the process by which programmers generate and evaluate hypotheses, and the manner in which programmers recursively attack a debugging problem. Unsupported are those complex debugging sequences involving multiple possible paths. This study extends the understanding of the mechanisms by which programmers choose, implement and evaluate debugging rules and strategies. Results verify that all observed programmers utilize a pattern of cognitive process steps as well as comprehension strategies to derive needed declarative knowledge. The results support a goal-driven model of program debuggers' cognitive processes depicting a hierarchy of levels with an inherent control structure that directs debugging process steps. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: software maintenance; problem solving; verbal protocol analysis; structural learning theory; expertise models; comprehension models; adaptive learning

1. INTRODUCTION

The number of controlled studies involving software programmers or analysts actively engaged in debugging has been lamented for over 20 years (Swanson, 1976; Sharpe, Haworth and Hale, 1991). Recently, Lieberman (1997, p. 28) stated, 'bordering on scandal is the fact that the computer science community as a whole has largely ignored the debugging problem', despite its complexity and cost. Although previous studies indicate that programmers spend between 50 and 90 per cent of

*Correspondence to: Professor Joanne E. Hale, Area of Management Information Systems, Department of Management Science and Statistics, Culverhouse College of Commerce and Business Administration, Box 870226, University of Alabama, Tuscaloosa AL 35487–0226, U.S.A. Email: jhale@cba.ua.edu

CCC 1040–550X/99/020073–19$17.50

Received 17 July 1998
Revised 15 January 1999

their debugging time comprehending the existing program (Robson *et al.*, 1991), these studies have not evaluated empirically how debuggers use this time based on a theoretically-grounded model.

In response to the need for greater understanding of the software debugging process, this work explores the cognitive processes used by programmers engaged in the software debugging process. To accomplish this goal, this study evaluates Hale and Haworth's (1991) proposed cognitive process model of debugging through the use of protocol analysis of professional COBOL programmers.

The underlying theoretical foundation of Hale and Haworth's model is human learning theory, specifically, structural learning theory (Jeeves and Greer, 1983; Scandura, 1977, 1984). Structural learning theory (SLT) explains how humans approach goal-driven cognitive processes by discovering and utilizing structure in the process steps they perform.

Hale and Haworth's model integrates both declarative knowledge (as articulated in Brooks' (1983) programmer comprehension model) and procedural knowledge (as explored in Gould's (1975) process model and Vessey's (1986) functional model). Information search goals and solution heuristic goals are explicitly included in their model. They develop a unified model of debugging knowledge and cognitive processes that is postulated to account for observed performance of both skilled and unskilled programmers. They state:

> 'The debugging model . . . allows for an iterative, strategy-switching process that proceeds from simple to difficult. The model also postulates a control mechanism for the debugging process. Thus the model accommodates strategies, semantics, and direct diagnostics in a fashion that suggests the reasons for performance differences . . . '
> (Hale and Haworth, 1991, p. 104).

However, prior to this study Hale and Haworth's model had not been empirically evaluated.

2. BACKGROUND

2.1. Three-level process with three elements

Hale and Haworth's model presumes a deterministic goal-orientated structured problem-solving approach that uses a multiple-level backward-chaining mechanism for goal management. Scandura (1977) demonstrated that children exhibit this goal-directed problem solving mechanism at an early age. Foshay (1988) extended the contentions to the use of the mechanism throughout an individual's life. Figure 1 depicts a simplified graphical representation of this cognitive approach, as presented by Hale and Haworth (1991). SLT postulates a three-level iterative process that relies on the execution of *rules*. All rules consist of three elements:

- the domain (characterizing the current undesirable state),
- the range (characterizing the desired goal state), and
- the transformation (the specific action necessary to move from domain to range).

2.2. First-level problem solving

Structural learning theory postulates that the first-level problem-solving process involves the execution of rules to directly eliminate a program bug. In debugging situations where the first-level process is sufficient to resolve the bug, the only process steps that will be invoked are *Find*

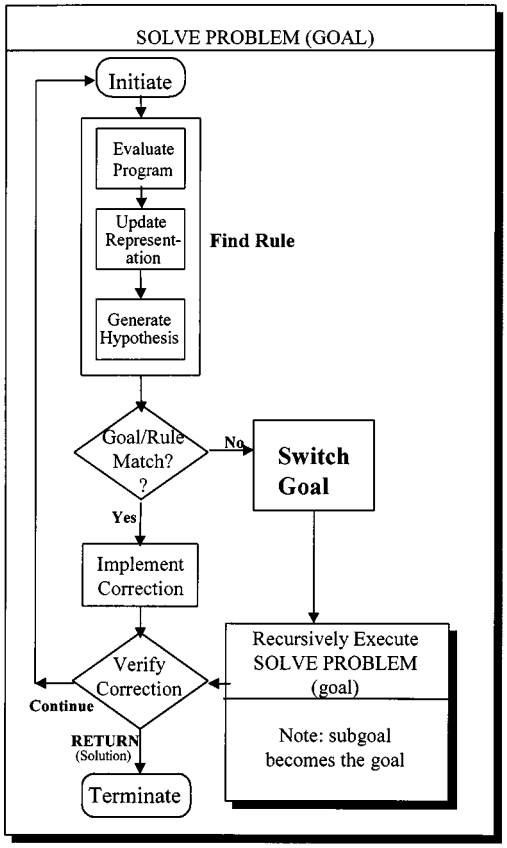

Figure 1. Debuggers' cognitive processes (adapted from Hale and Haworth (1991))

Rule, *Goal/Rule Match*, *Implement Correction* and *Verify Correction*. The *Find Rule* step structures and evaluates current assertions, formalizes this knowledge, generates a set of hypotheses that match domain (the undesired state) and range (the desired goal state), and attempts to find a transformation with the potential to satisfy the goal state, given the current domain state. The *Goal/Rule Match* step provides a branching operation; if a candidate rule exists that can transform the current state to the goal state, control is passed to *Implement Correction*. The *Implement Correction* step performs the necessary transformation; the *Verify Correction* step then evaluates the result to determine if the goal has been attained. If the goal has been met, the *Solve Problem* process successfully terminates. If the goal has not been met, control is transferred to *Find Rule* for another iteration.

2.3. Second-level problem solving

In situations where the *Find Rule* step fails to find a candidate direct solution rule to be executed, the model provides a second-level cognitive process analogous to backward chaining. Rather

than transferring control to *Implement Correction*, the *Goal/Rule Match* step branches control to the *Switch Goal* step. *Switch Goal* depicts the debugger defining a problem-solving strategy by establishing a secondary goal. The secondary goal is to gather new knowledge about the *Problem Situation* or about the primary *Goal* that will then allow the identification and application of a direct solution rule (at the first level of problem solving). The *Solve Problem* process is then recursively invoked using this secondary goal argument.

Hale and Haworth suggested that secondary goals are attained with the use of *action plans*. Action plans are rules that, when executed, may result in the attainment of additional knowledge about the *Problem Situation* or the primary *Goal*. The action plans used are program comprehension and bug location tactics or strategies (e.g., control-flow tracing, output discrepancy examination, slicing, etc.). The result of secondary goal attainment (successful obtainment of needed information regarding the current problem and goal states) is identified to the primary level as input to the *Solve Problem* activity, with the intent of identifying a direct solution rule capable of reaching the primary *Goal*. On completion of the recursively-invoked *Solve Problem* process, a check is made to determine if the original goal has been met. If the original goal has not been met, another iteration of the *Solve Problem* commences. Thus, this process iterates until the primary goal has been attained or all available action plans have been executed.

2.4. Third-level problem solving

A third problem-solving level is postulated by SLT, only executed when all known plans that could be invoked at level two have failed to provide the needed comprehension of problem and goal states. That is, upon failure to satisfy the primary goal and exhaustion of all available rules at the secondary level, the Hale and Haworth model suggests that programmers enter a tertiary problem-solving level. The operative goal at this level is identifying rules capable of creating an action plan to obtain additional information regarding the *Problem Situation* and the primary *Goal*, as input to the second level of problem solving. Following the execution of rules at the tertiary level, the newly created action plan is identified to the previous level for execution. This entire process of establishing new operative goals at higher levels of recursion continues until a direct solution rule capable of solving the primary *Goal* (successfully correcting a semantic program error) is developed and executed.

2.5. Adaptive nature of SLT

The knowledge gained through the execution of action plans provides the ability to develop new direct solution rules, thus developing debugging expertise. The proposed model depicts a debugging task as a function of the declarative knowledge possessed and utilized by an individual and controlled by the cognitive processes associated with the task. Upon failure to attain a direct solution to a debugging problem, the model characterizes the individual as switching to a higher-level goal to increase or clarify information about the goal or problem situation, which may lead to identification of a direct solution rule. The debugging process continues until the primary *Goal* (correction of a program bug) has been satisfied, until all candidate rules (including newly developed rules) have been evaluated, or the programmer cognitively thrashes through alternatives and eventually abandons the debugging process.

3. RESEARCH STUDY

3.1. Overview

Empirically evaluating Hale and Haworth's (1991) theoretical model requires studying the process used by programmers to debug programs. This is accomplished through a laboratory experiment during which subjects are asked to debug a program containing a hidden error while thinking aloud. The resulting verbal protocols are then coded to provide data, content analysis is used to validate the hypothesized process steps, and pattern analysis (specifically, lag-sequential analysis) is used to determine the relationships and patterns that exist between fundamental debugging process steps.

3.2. Research materials

To enable the generalization and comparisons across research studies, a COBOL sales reporting program was chosen that has been used in previous research studies (Vessey, 1986, 1989). The 570-line program in COBOL 85 was characterized as a highly structured and well-documented program. The individual program was unfamiliar to all of the research subjects, as is the case with many of the software maintenance tasks in industrial settings (Gibson and Senn, 1989; Shaft, 1995; Sheppard *et al.*, 1979). A single semantic error (removal of a trailing else statement) was introduced into the program. The research subjects received the source-code listing, a correct output report and an incorrect output report resulting from the execution of the program containing the error. The particular error was selected from a set of errors commonly found in practice (Gould and Drongowski, 1974), and considered moderately difficult to detect (Vessey, 1986). Because the error was only moderately difficult to detect, Hale and Haworth's third level of problem solving, creating a new action plan, was not tested. This level of problem solving is invoked only when an error is extremely difficult to detect, causing existing action plans to be unsuccessful in resolving the problem. Had the bug been too quickly detected, direct diagnosis could have been the only portion of the model assessed. Had the bug been too difficult to detect, programmers' cognitive thrashing could have been confounding.

3.3. Research subjects

20 professional programmers, all professionally trained and concurrently working in software maintenance positions, participated in the study. These participants had an average of five and a half years of information systems (IS) experience, and a range of three to five college-level programming courses. The mean elapsed time since they last engaged in software maintenance was 1.05 weeks, and 75% of the programmers engaged in a debugging activity the day of investigation. The job titles for the programmers ranged from entry-level programmers to senior programmer/analysts. Table 1 provides a demographic description of the experimental participants.

3.4. Experimental data gathering process

The experimental process centred on the 20 individual programmers resolving a semantic error in a COBOL sales reporting program. Prior to the primary data collection activity, each research subject:

Table 1. Descriptive statistics of research subjects

Characteristic	Unit	Mean	Standard deviation	Minimum	Maximum
Experience in IS position	Months	66.1	64.9	5	234
COBOL experience	Months	20.0	44.5	5	180
Formal programming training	Months	16.5	15.0	13	20
Time since last maintenance task	Weeks	1.0	2.7	0	12
Formal education	Years	16.5	1.5	13	19
Age	Years	30.8	6.7	21	45

- completed a background questionnaire,
- was briefed on the concurrent verbalization process,
- was given standard instructions detailing the debugging task to be performed, and
- debugged a short trial program to familiarize the subject with the concurrent verbal protocol process.

Subjects were instructed to verbalize their thought processes, and told that the study's administrator would give a neutral prompt (i.e., reminder) if nothing was said for seven seconds.

With this preparation, each subject was given the experimental program (consisting of the source code, incorrect output resulting from the source code and the desired output) to debug. The resulting debugging behaviour was collected using two unobtrusively placed audio–video cameras, which captured the subject's verbalized thoughts and movements through the documents.

3.5. Pilot study

Prior to primary data collection, a pilot study was conducted using two subjects. This pilot study served to ensure that the debugging task was capable of providing the data required, and to detect any possible problems in the data-collection process. Results of the pilot study and post-session interviews led to enhancements of the data-gathering process including:

- placement of the cameras and microphones,
- refinement of the trial problem, and
- enhancement of the programmer background survey.

3.6. Data preparation

The behaviour of each subject was captured on audio–video tape to provide data for verbal protocol coding. The recordings were transcribed, followed by breaking the verbalizations into segments (sentences, clauses or phrases), each representing one independent idea or action. These segments were then encoded. The adopted coding scheme (see Appendix) utilizes a predicate calculus notation in which predicates are used to represent postulated debugging process steps. The predicates are combined with their qualifying arguments using the functional notation $P(x, y, \ldots)$. Numerous researchers (Belkin, Brooks and Daniels, 1987; Diederich, Ruhmann and May, 1988;

Table 2. Encoder reliability measures

Reliability assessment type	Encoder(s)	Number of files	Mean (κ)	Standard deviation	Range
Intercoder reliability (κ)	Coder 1 × Coder 2	20	0.86	0.04	0.79–0.93
	Coder 1 × Coder 3	10	0.85	0.05	0.78–0.92
	Coder 2 × Coder 3	10	0.84	0.04	0.75–0.90
Intracoder reliability (κ)	Coder 1	3	0.86	0.03	0.82–0.88
	Coder 2	3	0.85	0.07	0.78–1.00
	Coder 3	2	0.86	0.02	0.84–0.87

Ericsson and Simon, 1984; Sanderson, James and Seidler, 1989) in verbal protocol encoding have used this general predicate approach.

Encoding protocols is a subjective manual operation, raising a number of theoretical and methodological questions regarding the objectivity and reproducibility of the encoding process (Nisbett and Wilson, 1977; Vessey, 1986). To address these concerns, this study used three trained, independent coders. Each coder encoded from the transcripts while referencing the audio–video tape for those semantics not adequately captured by the narratives. Two of the coders encoded all 20 subjects' behaviour, and the third encoded 10 of the subjects' behaviour. The encoding process yielded a minimum of 97 predicates for the subject finishing the exercise most quickly, to a maximum of 402 predicates for the subject taking the most time to complete the exercise.

The intercoder reliability measure adopted for this research is Cohen's kappa (κ). As detailed in Table 2, a global reliability measure of $\kappa = 0.85$ was obtained (representing the mean proportion of agreement between coders for the 10 commonly coded protocols), and the intercoder reliability ranged from 0.75 to 0.93. In comparison with other published IS work, this degree of agreement is quite acceptable (Belkin, Brooks and Daniels, 1987; Johnson, Zaulkernan and Garber, 1987; Shaft and Vessey, 1995; Vessey, 1986). Table 2 also depicts the individual intracoder reliabilities, as measured by the mean values of κ for protocol files re-coded by the same individual. These measures indicate an acceptable degree of consistency by the three coders.

3.7. Data analysis

3.7.1. Two forms

A comprehensive assessment of the research model is achieved by structuring the data analysis into two logical steps: *content* analysis and *pattern* analysis, as presented in Table 3. The content analysis provides an independent assessment of each of the postulated debugging process steps, and evaluates the presence of process steps not included in the *a priori* representation. The pattern analysis then examines the existence of patterns and relationships between the process steps distinguished in the content analysis. The distinction between the content and pattern analysis is consistent with the approach used in previous verbal protocol research (*vide* Biggs and Mock (1983), Byrne (1983), and Sanderson, James and Seidler (1989)).

Table 3. Summary of data analysis

Analysis step	Question addressed	Unit of analysis	Number of cases
Content	Do the hypothesized process steps accurately and completely describe debugging activities?	Encoded verbal segment (within research subject)	Ranges from 97 segments (subject 5) to 407 segments (subject 7)
Pattern			
within subject	Are the hypothesized process step sequences accurate for a given subject?	Process step pair (c, m) at lag l (within research subject)	Ranges from 94 step pairs (subject 5) to 401 step pairs (subject 7)
between subject	Are the process step sequences exhibited by the majority of programmers?	Subject	20

3.7.2. *Content analysis*

The intent of the content analysis is to determine if the process steps articulated in the research model (Figure 1) accurately and completely represent the debugging activities exhibited by the participants. This goal is accomplished by determining:

- if each of the proposed debugging process steps are represented by predicates derived from the subjects' verbalizations, and
- if there are observed subject verbalizations that are not captured by the proposed process steps.

The frequency with which predicates, and constants within predicate arguments, were used in the encoding of a specific protocol served to evaluate the postulated debugging process steps.

3.7.3. *Pattern analysis*

3.7.3.1. Two forms. Based on the results of the content analysis, pattern analysis is performed to assess the proposed process model. This analysis includes:

- a within-subject analysis, which identifies for each individual the process step sequences (at various sequential activity lags) that occur more often than expected at random; and
- a between-subject analysis, which identifies the process step sequences that are consistently observed across participants.

3.7.3.2. Within-subject analysis. The initial step in the pattern analysis is performed independently for each of the 20 participants. Within the protocol file of each subject, lag-sequential analysis,

a non-parametric variant of auto- and cross-correlation time series analysis techniques (Sackett, 1978), is used to evaluate sequential dependencies among observed process steps. Lag-sequential analysis does not directly identify exact patterns of occurrence among activities. However, it has notable advantages over alternatives. Compared with Markovian methods, there are less data to be dealt with at one time, without a necessary loss of information (Bakeman and Gottman, 1986; Sackett, 1978). Further, this technique provides sequential information unavailable from simple one-step contingency measures (Gottman and Roy, 1990), such as those used by von Mayrhauser, Vans and Howe (1997). The reader is referred to Sackett (1978) for a comprehensive discussion of the technique.

Lag-sequential analysis begins by designating one observed process step (A) as the criterion (c) and the remaining as matching process steps (m). The frequency with which the criterion process step is followed by each possible matching step after $(\text{lag}\,(l) - 1)$ intervening steps is analysed to determine the strength of lagged-sequential dependencies. Specifically, the null hypothesis, where t indicates a time

$$H_0 : P(A_{t+l} = m \mid A_t = c) \leq P(A_{t+l} = m) \tag{1}$$

is tested against the alternative hypothesis

$$H_1 : P(A_{t+l} = m \mid A_t = c) > P(A_{t+l} = m) \tag{2}$$

To test the null hypothesis of process-step independence, the test statistic Z (Bakeman and Gottman, 1986; Gottman and Roy, 1990) is calculated as follows:

$$Z_{smcl} = \frac{\hat{p}(A_{t+l} = m \mid A_t = c) - \hat{p}(A_{t+l} = m)}{\sqrt{\dfrac{\hat{p}(A_{t+l} = m)(1 - \hat{p}(A_{t+l} = m))}{(P - l)\,\hat{p}(A_t = c)}}} \tag{3}$$

where:

$\hat{p}(.)$ = the observed probability of (.),
s = the designated subject,
m = the designated matching process step,
c = the designated criterion process step,
t = a designated time,
l = the designated time lag,
A_t = the process step exhibited by subject s at time t, and
P = the total number of predicates recorded for the subject s.

For each possible combination of subject s, criterion process step c, matching process step m and lag l, Z tests if, for subject s, process step m follows process step c after $(l - 1)$ intervening steps significantly more frequently than would occur at random. In turn, each predicate in the encoding scheme is designated as the criterion c. A probability profile is developed to depict the interaction of each process step in relation to all of the possible matching process steps for a specific lag l and research subject s. The result of the within-subject analysis is a set of statistically significant lagged process step pairs found at the subject level (using $\alpha = 0.05$, as recommended by Bakeman and Gottman (1986, pp. 143–145)).

3.7.3.3. Between-subject analysis. Results of the within-subject analysis are then used as input to the between-subject analysis, in which a frequency count of each individually significant lagged process step pairs is made for the 20 subjects. The number of subjects for which a specific lagged process step pair is significant is a binomial variable, with parameters $n = 20$ and p. Then a binomial test is conducted for each of the process step pairs to identify those that are consistently observed. Specifically, the null hypothesis

$$H_0 : p \leq 0.5 \tag{4}$$

is tested against the alternative hypothesis

$$H_1 : p \geq 0.5 \tag{5}$$

Rejection of the null hypothesis leads to the conclusion that the majority of programmers from the sampled population will consistently exhibit the tested lagged process step pair.

3.8. Results

The content analysis, summarized in Table 4, supports a final descriptive model that includes each of the debugging process steps identified in the research model. Each process step derived from the research model is consistently performed by each of the 20 programmers. In addition, the content analysis does not support the inclusion of any additional steps into a descriptive software-debugging model. Thus, it can be concluded that the individual debugging process steps proposed by the Hale and Haworth model adequately describe the actual debugging activities exhibited by the research subjects.

Based on the confirmatory results of the content analysis, pattern analysis is conducted. Table 5 depicts the results of this analysis and shows:

- the lagged process step pairs consistently observed across the research group;
- the proportion of subjects exhibiting the significant relationship between the process steps; and
- the critical level resulting from the binomial tests.

4. DISCUSSION

Figure 2 maps the content analysis (Table 4) and process analysis (Table 5) to Hale and Haworth's proposed research model (Figure 1). Focusing on the content analysis, Figure 2 reveals that the set of hypothesized process steps may be concluded as both:

- accurate (all process steps were exhibited by all subjects) and
- complete (no additional process steps were exhibited by the subjects)

in its representation of the tasks encompassed by the debugging process.

Focusing on the process analysis, Figure 2 reveals that the hypothesized sequence by which the debugging process steps are followed is only partially supported. Based on the binomial tests

Table 4. Content analysis of observed process steps

Process step	Percentage of participants demonstrating step	Mean percentage of total activities
Evaluate Program	20/20	32.2%
Update Representation	20/20	30.0%
Generate hypothesis	20/20	5.6%
Goal/Rule Match	20/20	1.0%
Switch Goal	20/20	21.5%
Implement Correction	20/20	1.1%
Verify Correction	20/20	5.1%
Create/modify search strategy[1]	2/20	0.1%
No code for activity[2]	6/20	0.4%
Silence/prompts[3]	14/20	1.6%
Non-productive[4]	13/20	1.5%

[1]This process step represents Hale and Haworth's (1991) level-three problem-solving process, which was not tested by this study.

[2]Non-codeable behaviours were found to be attempts to interact with the monitor during the debugging task for the purpose of posing a task-related question, or verbalizing a physical activity facilitating the debugging task (e.g., jotting a note). Each is a manifestation of the verbal protocol method or the manual desk-checking procedure rather than a process step that discredits the proposed model.

[3]The underlying process step associated with a period of silence can only be inferred, but an examination of the associated transition frequency matrices reveals that approximately 74.5% of these segments follow either an *Evaluate Program* or an *Update Representation* process step. The neutral prompt administered by the researcher is often followed by a restatement of the process step immediately preceding the prompt and an explanation of the purpose for performing this step. Based upon this observation, it is believed that the periods of silence are not sufficient to invalidate the postulated software debugging representation.

[4]The non-productive protocol segments were found to be comments directed toward:

- debugging source code written by someone else;
- the lack of automated support for variable location, calculations and program change verifications; and
- initial discomfort with concurrent verbalizations early in the debugging process.

These factors have been previously acknowledged in the literature as objections to the use of manual desk-checking and concurrent verbalization; however, these objections are offset by increased control and data richness derived from the use of the manual debugging procedure. The limitations are not believed to have impaired the validity of the resulting data.

conducted, it is valid to conclude that several hypothesized sequence pairs (five of the eight hypothesized sequence pairs) accurately portray the debugging process followed by the majority of programmers.

The proposed research model effectively represents the process by which programmers find candidate rules. The majority of programmers consistently search for rules with the potential to satisfy the goal state by:

- first evaluating the program (determining known facts and assertions regarding the problem situation); then

Table 5. Observed process step pairs noted across the research group

Lag	Process step pair	Observed proportion	Critical level
	Hypothesized		
Lag 1	Evaluate Program → Update Representation	19/20	$p < 0.00001$
	Update Representation → Generate Hypothesis	15/20	$p < 0.05$
	Generate Hypothesis → Goal/Rule Match	14/20	$p < 0.06$
	Implement Correction → Verify Correction	14/20	$p < 0.06$
	Switch Goal → Evaluate Program[1]	18/20	$p < 0.0005$
	Un-hypothesized		
	Switch Goal → Implement Correction	15/20	$p < 0.05$
	Hypothesized		
Lag 2	Generate Hypothesis → X → Switch Goal	14/20	$p < 0.06$
	Switch Goal → X → Update Representation	18/20	$p < 0.0005$

[1]The recursive execution of the entire *Solve Problem* process that is graphically represented in Figure 1 is implied in the research model. No codeable verbal protocol predicate is used to define the recursive process. Rather, a linkage between the cognitive process tasks of *Switch Goal* and *Evaluate Program* implies the recursive execution.

- updating the program's representation (structuring those facts and assertions); and finally
- generating an hypothesis (making further assertions about the problem situation, such as program functions and variables, or bug affiliations and locations).

Further supported was the manner in which programmers tend to attack recursively a debugging problem. Participants consistently followed a goal switch (the purpose of which is to gather further information regarding the problem or primary goal) with re-entering the *Solve Problem* process by re-evaluating the problem. This finding of a consistently employed control mechanism is a critical addition to the debugging literature. Further research is needed to explore the complexities of this previously ignored control mechanism.

The research model was unsuccessful in representing other aspects of the debugging process. Specifically, the three hypothesized but unsupported process step sequences are:

- goal/rule match to switch goal;
- goal/rule match to implement correction; and
- verify correction to evaluate program.

These three sequences have two common characteristics:

- all originate from a branch decision point, at which a debugger has multiple possible paths; and
- all involve a process step that represents a relatively small portion of the programmers' time (goal/rule match: 1%; verify correction: 5%).

Thus, the lack of empirical support for these hypothesized sequences may arise from a lack of understanding regarding the branch decision points, errors in coding these decision points and/or

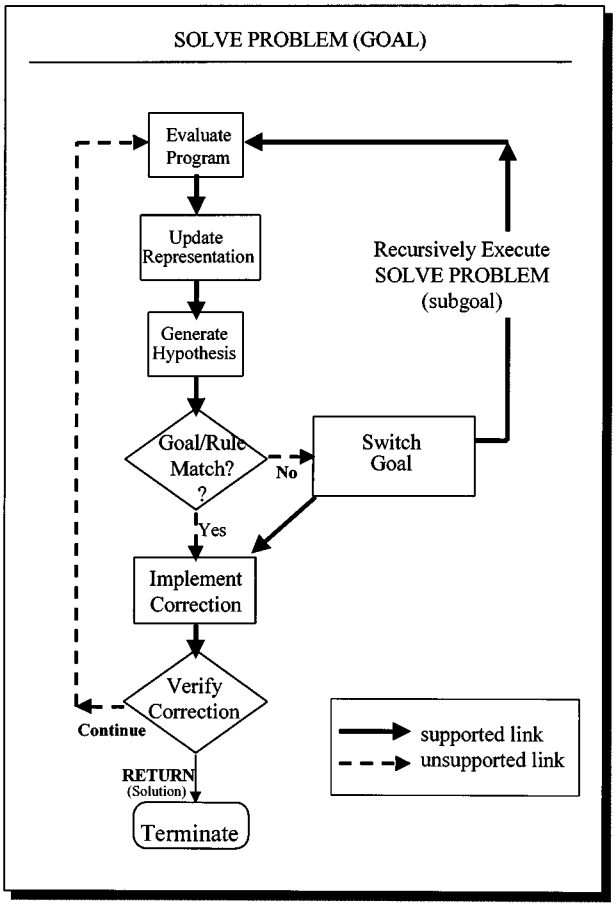

Figure 2. Content and process analysis linked to the research model

the lack of statistical power resulting from the small amount of data. Further research is required to determine which of these factors are responsible for the lack of model fit.

A further weakness in the research model is found in the single unhypothesized sequence (switch goal to implement correction) consistently observed across the group. It is hypothesized that this sequence occurs as a programmer becomes frustrated with an inability to find a rule that matches the current goal. This induces cognitive thrashing, characterized by seemingly random rule execution with no application of declarative or process knowledge related to previous goal-orientated behaviours. It is believed that to cope with this cognitive thrashing, the programmer switches to a simpler, more attainable goal (e.g., understanding the desired program output). The recursive execution of *Solve Problem* is actually implemented, but is done so rapidly and intuitively (due to the goal's simplicity) that it is not verbalized. Although further research is needed to determine the validity of this hypothesis, it is anecdotally supported by a review of the audio–video tapes of several participants.

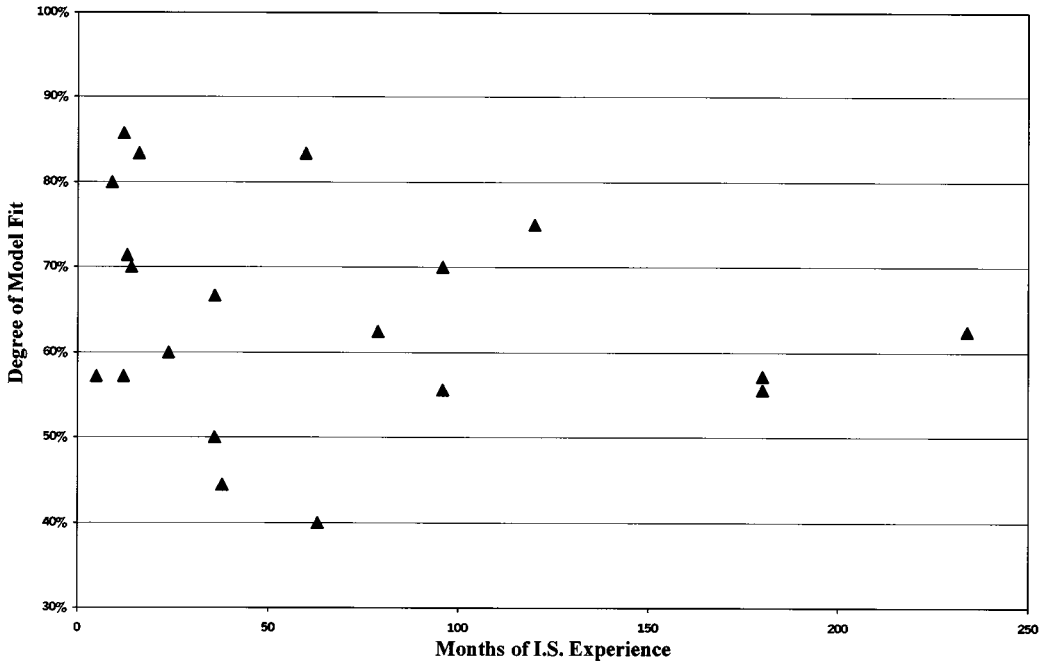

Figure 3. Scatter plot of the subjects' level of experience and fit with the model

An important aspect of the proposed research model is its claimed broad compatibility across programmers. According to Hale and Haworth (1991), the proposed process model holds across levels of programmer expertise. Although there are insufficient data to test this claim statistically, a scatter plot depicting the 20 (level of experience, model fit) pairs can provide valuable anecdotal insight into this aspect of the proposed model. An individual's level of experience is measured by the total number of months served in any IS-related position, and ranges from 5 months to 234 months. For each subject, the degree of model fit, $Fit_{s..1}$, is measured by the ratio

$$Fit_{s..1} = \frac{\sum_{all(m,c)} p \times h}{\sum_{all(m,c)} p} \tag{6}$$

where:

$p = 1$ if the lag 1 process step pair (m, c) is significant for subject s and 0 otherwise,
$h = 1$ if the lag 1 process step pair (m, c) is as hypothesized by the research model and 0 otherwise.

$Fit_{s..1}$ ranges from a low of 40% to a high of 86%. This measure represents the percentage of a participant's significant lag 1 process step pairs in support of the research model. Thus, model fit is reduced by either a decrease in the number of significant hypothesized process step pairs or an increase in the number of significant non-hypothesized process step repairs. Shown in Figure 3, this scatter plot shows no apparent relationship between degree of model fit and experience level.

Figure 3 reveals that the research model's ability to describe individual debugging processes accurately and completely ranged from 40% to 87%. The model's fit for the majority of subjects is above 60%, and the number of months of information systems experience did not account for differences in the model's descriptive ability.

5. CONCLUSIONS

The sequence by which programmers of all experience levels perform debugging activities is better understood as a result of this study. This study has

- revealed the consistent process by which programmers search for suitable rules to invoke;
- revealed the recursive manner in which programmers doing debugging attack problems for which a solution cannot be diagnosed directly; and
- contributed to the software maintenance field by showing that programmers naturally engage in the hypothesized set of debugging activities.

Recognizing and understanding this common set of activities can improve software debugging training and support.

The SLT model and the empirical results of this study indicate that debugging training programs should include techniques to:

- structure the problem situation including describing the goal, the type of error and known assertions;
- maintain the knowledge gained throughout the debugging episode so that known facts and assertions are readily available;
- develop diagnoses based on desired outcomes and situation states;
- develop plans to transform current states to desired outcomes based on the diagnoses;
- gain additional problem situation knowledge through a repertoire of comprehension strategies; and
- effectively execute the control mechanisms of at least the first two levels of the three-level SLT process model.

Thus, this study makes evident the need to structure the training of software programmers and analysts to be aligned with the process of SLT, which emphasizes comprehension strategies, situational variables that dictate the application of strategies and the formation of hypotheses.

This study failed to fully support the study's research model. In particular an additional link between the activities *Switch Goal* and *Implement Correction* was found that was not predicted by SLT. This new link may indicate a needed modification in the debugging process model or may indicate a limitation in the collection capability of verbal protocol analysis. The latter is caused by relying solely on the audible words of participating subjects rather than on the interpretation of behaviour. Verbal protocol analysis is inadequate in detailing either purely cognitive activities that do not result in a spoken word or 'chunked' cognitive activities that result in only a single verbal output.

For example, this new link might be explained by returning to the *Switch Goal* activity from a level-two comprehension session that produces more situational knowledge than can be used to

invoke a rule. The cognitive activities of *Find Rule* and *Goal/Rule Match* are clustered together so tightly with *Implement Rule* that the only audible words coincide with the *Implement Rule* activity. More research is needed to determine the factors causing the non-hypothesized sequence and to refine the model for improved robustness.

The results of this study strongly support the Hale and Haworth model in a COBOL environment with a bug moderately difficult to detect. More experimentation is needed to investigate debugging situations in which software programmers and analysts must establish new comprehension strategies when none exist, and to explore how they prioritize sets of rules, hypotheses and comprehension strategies when more than one may satisfy the goal state.

APPENDIX: SOFTWARE DEBUGGING ENCODING SCHEME

Predicate	*Argument(s)*	*Notes*
Implement correction	(<Action>,<Affected>)	'Action' is: insert/statement delete variable/statement change variable/statement 'Affected' is: program function/variable bug location/affiliation
Generate Hypothesis	(<About>)	'About' is: program functions program variables bug location bug affiliations
Evaluate	(<What>,<Strategy>)	'What' is: code statements program comments output (correct/incorrect)
Goal	(<What>,<Level>)	'What' is: correct program comprehend/locate create plan to compute/locate
Select Strategy	(<To>)	Strategies: backward search causal reasoning control flow examine output simulate execution slice not recognizable

Update Representation	(<Of>)	'Of' is: program changes program representation
Verify	(<What>)	'What' is: program changes program representation
Non-Productive	(<Manner>)	'Manner' is: discomfort with situation distracted
Non-Codeable	(<Why>)	'Why' is: silence/prompts no code for step

References

Bakeman, R. and Gottman, J. M. (1986) *Observing Interaction: An Introduction to Sequential Analysis*, Cambridge University Press, New York NY, 221 pp.

Belkin, N. J., Brooks, H. M. and Daniels, P. J. (1987) 'Knowledge elicitation using discourse analysis', *International Journal of Man–Machine Studies*, **27**(2), 127–144.

Biggs, S. F. and Mock, T. J. (1983) 'An investigation of auditor decision processes in the evaluation of internal controls and audit scope decisions', *Journal of Accounting Research*, **21**(1), 234–255.

Brooks, R. (1983) 'Towards a theory of comprehension of computer programs', *International Journal of Man–Machine Studies*, **18**(6), 543–554.

Byrne, R. (1983) 'Protocol analysis in problem solving', in Evans, J. S. B. T. (Ed), *Thinking and Reasoning: Psychological Approaches*, Routledge and Kegan Paul, London, pp. 227–249.

Diederich, J., Ruhmann, I. and May, M. (1988) 'A knowledge-acquisition tool for expert systems', *International Journal of Man–Machine Studies*, **26**(1), 29–40.

Ericsson, K. A. and Simon, H. A. (1984) *Protocol Analysis*, MIT Press, Cambridge MA, 426 pp.

Fisher, C. and Sanderson, P. (1996) 'Exploratory sequential data analysis: exploring continuous observational data', *Interactions*, **3**(2), 25–34.

Foshay, R. (1988) 'I don't know is on third', *Performance & Instruction*, **27**(9), 8–19.

Gibson, V. R. and Senn, J. A. (1989) 'System structure and software maintenance performance', *Communications of the ACM*, **32**(3), 347–358.

Gottman, J. M. and Roy, A. K. (1990) *Sequential Analysis: A Guide for Behavioral Researchers*, Cambridge University Press, New York NY, 275 pp.

Gould, J. D. (1975) 'Some psychological evidence on how people debug computer programs', *International Journal of Man–Machine Studies*, **7**(2), 151–182.

Gould, J. D. and Drongowski, P. (1974) 'An exploratory study of computer program debugging', *Human Factors*, **16**(3), 258–277.

Hale, D. P. and Haworth, D. A. (1991) 'Toward a model of programmers' cognitive processes in software maintenance: a structural learning theory approach for debugging', *Journal of Software Maintenance*, **3**(2), 85–106.

Jeeves, M. A. and Greer, G. B. (1983) *Analysis of Structural Learning*, Academic Press, New York NY, 269 pp.

Johnson, P. E., Zaulkernan, I. and Garber, S. (1987) 'Specification of expertise', *International Journal of Man–Machine Studies*, **26**(2), 161–181.

Lieberman, H. (1997) 'The debugging scandal and what to do about it', *Communications of the ACM*, **40**(4), 27–29.

Nisbett, R. E. and Wilson, T. D. (1977) 'Telling more than we can know: verbal reports on mental processes', *Psychological Review*, **84**(3), 231–259.

Robson, D., Bennett, K. H., Cornelius, B. J. and Munro, M. (1991) 'Approaches to program comprehension', *The Journal of Systems Software*, **14**(2), 79–84.
Sackett, G. P. (1978) 'Measurement in observational research', in Sackett, G. P. (Ed), *Observing Behavior: Data Collection and Analysis Methods, Volume II*, University Park Press, Baltimore MD, pp. 25–43.
Sanderson, P. M., James, J. M. and Seidler, K. S. (1989) 'SHAPA: an interactive software environment for protocol analysis', *Ergonomics*, **32**(11), 1271–1302.
Scandura, J. M. (1977) *Problem Solving*, Academic Press, New York NY, 591 pp.
Scandura, J. M. (1984) 'Structural (cognitive task) analysis: a method for analyzing content. Part II: toward precision, objectivity, and systematization', *Journal of Structural Learning*, **8**(1), 1–28.
Shaft, T. M. (1995) 'Helping programmers understand computer programs: the use of metacognition', *ACM SIGMIS Database*, **26**(4), 25.
Shaft, T. M. and Vessey, I. (1995) 'The relevance of application domain knowledge: the case of computer program comprehension', *Information Systems Research*, **6**(3), 286–298.
Sharpe, S., Haworth, D. A. and Hale, D. (1991) 'Characteristics of empirical software maintenance studies: 1980–1989', *Journal of Software Maintenance*, **3**(1), 1–15.
Sheppard, S., Curtis, B., Milliman, P. and Love, T. (1979) 'Modern coding practices and programmer performance', *IEEE Computer*, **12**(12), 41–49.
Swanson, E. B. (1976) 'The dimensions of maintenance', in *Proceedings Second International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 492–497.
von Mayrhauser, A., Vans, A. M. and Howe, A. E. (1997) 'Program understanding behaviour during enhancement of large-scale software', *Journal of Software Maintenance*, **9**(5), 299–327.
Vessey, I. (1986) 'Expertise in debugging computer programs: an analysis of the content of verbal protocols', *IEEE Transactions on Systems, Man, and Cybernetics*, **16**(5), 621–637.
Vessey, I. (1989) 'Toward a theory of computer program bugs: an empirical test', *International Journal of Man–Machine Studies*, **30**(1), 23–46.

Authors' biographies:

Joanne E. Hale is a faculty member in the Area of Management Information Systems in the Culverhouse College of Commerce and Business Administration at the University of Alabama. She holds a Ph.D. in MIS from Texas Technical University, as well as an MA in Statistics and BS in Industrial Engineering from the University of Missouri. Her research interests include software maintenance, information systems for the support of crisis management, component-based development and IS strategic alignment. Email: jhale@cba.ua.edu

Shane Sharpe is a faculty member in the Area of Management Information Systems in the Culverhouse College of Commerce and Business Administration at the University of Alabama. He received his Ph.D. in MIS from Texas Technical University. His current research interests include component-based software development methods, software maintenance and IT support for enterprise integration. Email: ssharpe@cba.ua.edu

David P. Hale is a faculty member in the Area of Management Information Systems in the Culverhouse College of Commerce and Business Administration at the University of Alabama. He received his Ph.D. in MIS from the University of Wisconsin–Milwaukee. He is Director of the Alabama Enterprise Integration Laboratory and the Center for Manufacturing Information Technology where his current research interests include software maintenance, component-based software development, methods modelling, software reuse and IT architecture. Email: dhale@cba.ua.edu

